Graph-based Simultaneous Localization and Mapping on the TurtleBot platform

Rik Claessens, Yannick Müller, Benjamin Schnieders

January 23, 2013

Abstract

Simultaneous localization and mapping (SLAM) is a non-trivial task in robotics. SLAM is an algorithm that generates a map of an environment that is previously unknown to the robot while at the same tracking the location of the robot within that environment. Using the TurtleBot platform and the Robot Operating System (ROS), a Graph-based SLAM algorithm (GraphSLAM) is successfully implemented and discussed. Additionally, an exploration strategy is implemented to allow the robot to explore and map the environment autonomously.

1 Introduction

SLAM is a technique used in robotics to build a map of an unknown environment without prior knowledge while at the same time localizing the robot relative to its surroundings. These two tasks cannot be performed individually. Feedback from the map is necessary to localize the robot accurately and the position of the robot is utilized when updating the map. Both aspects are error prone. The SLAM algorithm handles this iterative process of map and location updates and aims to overcome the mutual dependency.

It is clear that the robot needs to move through the environment in order to generate a map. To achieve this, the robot can either be controlled directly or by an exploration strategy. An exploration strategy determines the next goal for the robot to navigate to.

For this article, the Robot Operating System (ROS) was used on the TurtleBot platform to implement a GraphSLAM algorithm. In Section 2, the environment the robot will be operating in are explained. After this, Section 3 explains the notion of SLAM and introduce severals implementations of SLAM algorithms. Section 4 describes the components required for an exploration strategy. The implementation of all these components is outlines in Section 6. In Section 7 the conducted experiments and the results are discussed. Finally, Section 8 contains several conclusions and poses a number of future research questions.

2 Environment

In this section the operating environment for the robot is described. First ROS is discussed, afterwards the TurtleBot platform is explained in detail.

2.1 Robot Operating System

ROS [12] is an open-source software framework that serves like an operating system for robots. While originally developed by the Stanford Artificial Intelligence Laboratory under the name *switchyard*, it is currently actively being developed by Willow Garage, a company focused on stimulating research on robotics by providing the ROS framework.

The ROS software runs on Ubuntu Linux (other operating systems are still ‘experimental’), and provides, besides many software packages, low-level access to robotic devices. ROS is a tools-based, distributed software framework, in which distinct nodes communicate using a shared messaging layer. This messaging service allows nodes built using different programming languages to communicate with each other, external debugging, visualizing data or recording and playing back certain scenarios. ROS also provides the simulation package Stage, which allows to visualize all kinds of data regarding the robot and its environment.

The ROS framework is updated often, both iterative updates as well as major releases. Because the newest version of ROS, called *Groovy Galapagos*, was released during the development of the SLAM algorithm described in this article the previous version of ROS was used, *ROS Fuerte*, released in April 2012. Because ROS is aimed at the open source community, bugs get fixed very fast and new features are added regularly.

2.2 OpenCV

OpenCV is an open source library designed for real-time computer vision and image processing. Developed by Intel, it is now supported by Willow Garage. Among others, it's main features are motion tracking, image segmentation and object recognition. For the latter, multiple trainable classifiers exist. Object recognition is used in this article for landmark detection.

2.3 TurtleBot

The TurtleBot is an open hardware project, based on the iRobot Create platform, presented by Willow Garage in 2011. In the most basic form, it is a wheeled robot controlled by an Asus EeePC 1215N with an Intel® Atom^{TM} D525 Dual Core Processor. Odometry can be improved by using a single axis gyroscope available with the robot. The TurtleBot used for this article was furthermore equipped with an Hokuyo URG-04LX-UG01 laser range finder, which was used to map the environment.

The TurtleBot platform can be extended with a Microsoft Kinect^{TM} camera, which gives it the ability to produce 3D-images. This article however, is only concerned with 2D sensor data and 2D maps.

3 SLAM

As described in Section 1, SLAM algorithms deal with the iterative process of map and robot configuration updates. The robot configuration is normally described by six variables: the three-dimensional Cartesian coordinates describing the position of the robot in its environment, and the robots three Euler angles, roll, pitch and yaw. While this article only focuses on 2D-maps, the robot configuration can be described with only three variables: the robots pose (Equation 1), which is described by two-dimensional planar coordinates, and the robots heading or angular orientation.

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{1}$$

There is no definitive solution for the SLAM problem, because of a lot of different factors. There are a numbers of aspects about robot exploration that are error prone or hinder the performance of the robot in another way. First, the robots sensors will never be 100% accurate. Furthermore, the robots motors are error prone, so the odometry data obtained from the robot does not suffice to determine the robots location or even its movement. Another aspect which makes the problem much more complex are dynamic environments. Since the robot does not have prior knowledge of the environment it is located in, a change in an already explored part of the environment might decrease the performance drastically.

For all the problems mentioned above there are a number of strategies to approximate a solution. In the following subsections different strategies and their (dis)advantages are explained for each aspect. In the end of this section a number of different SLAM algorithms are discussed.

3.1 Robot movement

Due to either control noise or environmental influences, the result of an action performed by the robot is uncertain. However, to accurately build a map of the environment, the pose - the coordinates and direction - of the robot needs to predicted as well as possible.

For modeling the movement of the robot and predicting its pose, a good estimation is needed for the robots pose after an action. A probabilistic kinematic model aims to predict the posterior pose of a robot given a control input. This conditional probability is given by Equation 2.

$$p(x_t | u_t, x_{t-1}) \tag{2}$$

In this equation x_{t-1} and x_t represent the pose of the robot at time $t-1$ and t respectively and u_t represents the control input of the robot. Equation 2 describes the probability of the robot ending up in kinematic state x_t, when executing u_t when the robot is in state x_{t-1}.

There are two commonly used kinematic models in robotics, the velocity motion and the odometry motion model. Since the TurtleBot platform is not equipped with an acceleration sensor, but only with odometry sensors, only the odometry motion model can be used.

Odometry motion model

Another motion model is the odometry motion model. This model incorporates odometry measurements of the robot when calculating posteriors. This odometry data is obtained by sensors which measure the movement of the robot. This can be a disadvantage, since this data will only be available after the robot has moved. In comparison, the velocity motion model can be used before the robot has moved, as it performs a forward simulation of the robots movement. For mapping localizing the robot or mapping its environment this poses no problem because they do not require motion planning.

Strictly speaking, odometry data are sensor measurements, not controls. In order to reduce the size of the state space in the motion model, this data is considered just like controls of the velocity motion model. Each control u_t of the odometry motion model can be described by Equation 3.

$$u_t = \begin{pmatrix} \overline{x}_{t-1} \\ \overline{x}_t \end{pmatrix} \tag{3}$$

The transition from x_t to x_{t-1} can be split into three actions. The first action is a rotation δ_{rot1}, followed by a straight line motion δ_{trans} and finally another rotation δ_{rot2}. This is depicted in Figure 1.

The odometry motion model assumes that there exists noise for each action separately. As was the case for the velocity motion model, sampling requires a lot less resources than calculating an accurate density function over x_t.

As is the case for both motion models, errors in early stages of the model will cause the approximation of the real situation to become less and less accurate as time goes by and the robots executes more actions. In order to reduce this error, data from the robots sensors can be used. This is described in the next section.

3.2 Robot sensors

The TurtleBot platform is equipped with a laser sensor which has a scan window of 240°. Its range is four meters, with an accuracy of 97% for distances greater than one meter, and a deviation of ± 30mm for distances closer than one meter.

Another influence on the robot sensors is the environment. Different surfaces reflect the lasers of the TurtleBots laser sensor in different ways. Very dark surfaces or glass surfaces may not reflect the laser at all, while others will ricochet the laser beam away from the robot (surfaces like mirrors).

In order to cope with these issues, a measurement model is needed.

Scan Matching

In order to detect loop closure and to improve odometry estimations immediately, the recent scan can be matched against former scans. Input data for scan matching is the pose estimate retrieved from odometry, and the raw laser distance measurements. Scan matching should return an updated transformation between the two scans, as well as a measure of confidence for the corrected transformation. The correction should only be applied if the confidence is above a certain threshold, otherwise the scans did possibly not overlap.

3.3 Robot environment

For this article the performance of the algorithm was tested in the Maastricht University Swarmlab (Figure 2). In this room there are a number of glass doors which do not reflect the beams from the robots laser sensor. Also there are a number of desks and chair which the robot is not able to detect. These rather simple problems pose a problem for a robot when it is required that it should function in all circumstances. However, this was not the scope of this article and thus, during the tests, the robot is manually corrected when encountering one of the situations mentioned above.

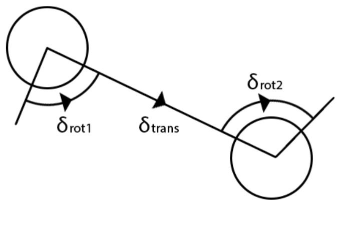

Figure 1: The odometry motion model

Another problem which a robot could face in an environment is the so-called ‘Robot kidnapping problem’. The robot could be moved by an external source while it is unable to track this movement. This could happen when somebody lifts the robot up and puts it down somewhere else. The robot has to localize itself again when it is moved, which could lead to incorrect map updates.

To compensate for that, addition of new nodes should only be done if the scan matching algorithm finds some correspondence. If it does not, the robot should explore locally until the current scan matches an old one, and continue exploration for that. A particle filter may improve the localization of the robot, as it gives an expectancy of the robot’s position. Also, the map itself can be queried for distinct features (such as corners) and the robot steered towards them, in order to speed up recovery.

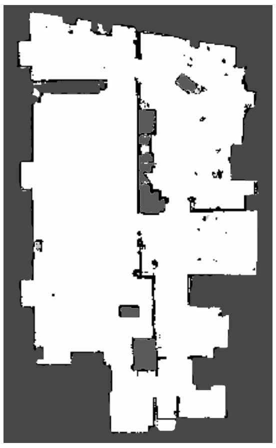

Figure 2: Maastricht University Swarmlab

3.4 SLAM Algorithms

In this subsection several SLAM algorithms are introduced. First, the intuition of the algorithm is explained, then the way in which they deal with the issues mentioned in the preceding subsections is discussed. Finally their (dis)advantages are described in detail.

ROS GMapping

During the first phase of the project the goal was to implement an exploration algorithm. In order to test it, the ROS GMapping package was used as the SLAM algorithm. ROS GMapping is a SLAM implementation provided by ROS and requires no knowledge of actual SLAM algorithms in order to be used.

ROS GMapping uses a particle filter to represent possible maps. To reduce the number of particles, both the movement and measurement of the robot needs to be taken into account. First, the robot is trajectory is estimated by one particle. Then, using this robot trajectory the map is estimated for each particle. For all particles their 'weight' is calculated and the collection of samples will be resampled to represent the real environment as well as possible.

GraphSLAM

GraphSLAM is SLAM algorithm which uses a graph built from constraints accumulated while exploring the environment. The algorithm calculates a map posterior by linearizing this set of constraints. The set of constraints is built from robot positions and map features, called motion arcs and measurement arcs respectively. Motion arcs connect two robot positions and a measurement arc connects a robot position with one measurement in that position. Every arc in the graph corresponds to a nonlinear constraint.

While a robot is exploring its environment, the size of the graph will grow with new motion and measurement arcs. This action is cheap as it just adds several new constraints to the system. However, to obtain a full map posterior, each constraint needs to be considered. Therefore GraphSLAM is no incremental algorithm, but a so-called *lazy SLAM* algorithm. An advantage of this is that GraphSLAM is able to apply several improvement techniques like data association and reconsideration of previous calculations whenever new information becomes available.

GraphSLAM is an iterative algorithm in which three main steps form the basis: constructing the map, calculating correspondence variables and linearizing the motion and measurement models. The result is the best estimate of all three quantities, which delivers a map which are often more accurate than other SLAM algorithms produce. This accuracy comes at the cost of the requires resources in comparison to other SLAM implementations.

4 Exploration

If the robot's sensors covered the whole environment, exploration would be unnecessary. However, most sensors have a limited range and obstacles present in the environment block sensor readings. Exploration is a technique to move a robot to all positions needed to observe every feature of the environment at least once.

Exploration strategies can be divided in two types, uninformed exploration and informed exploration. Uninformed exploration is a search strategy that steers the robot along a fixed pattern. Informed exploration is a adaptive to the environment the robot is situated in and incrementally adds sensor information to the knowledge of the robot. Exploration performance is usually measured by the time it takes a robot to fully explore a certain environment.

In this section different ways to explore are discussed. These includes frontier detection algorithms and several different exploration strategies.

4.1 Uninformed Exploration

A common example for uninformed exploration is wall following, an easily implementable maze solving algorithm. A common problem with just following a wall is enclosed loops in the environment. A simple, yet effective way to avoid getting stuck in loops is to randomly switch sides with a low probability. However, it is clear that, while they are easy to implement and fast to execute, uninformed exploration strategies will always have limitations in complex environments. Therefore, in the next section, a more sophisticated exploration strategy is discussed.

4.2 Frontier Detection

Using knowledge when it is available produces almost always better results than using a fixed strategy in any domain. This is also the case for the exploration strategy of a robot executing a SLAM algorithm. While revisiting an area can improve map quality, it is not useful to explore one area, while ignoring other areas of the environment. Instead it would be better to consider which areas are already explored and explore not yet discovered parts of the environment.

In order to achieve this, the robot needs a way of detecting where unexplored areas are located and a strategy that sets one of these areas as a goal to explore next. For the first problem, frontier detection is implemented.

Frontier detection is an algorithm which detects the borders between explored and unexplored areas of the map, called frontiers. When the robot will travel to a frontier is guaranteed to gain new information, as it has not yet made any measurements at this location. Upon arriving at a frontier, new information about the environment will become available, which will be incorporated in the map by the SLAM algorithm. Then the frontier detection algorithm computes a set of new frontiers. This step reduces the (in this case) two dimensional map to a one dimensional set of frontiers.

There are two commonly used implementations of frontier detection, Wavefront Frontier Detection (WFD) of Fast Frontier Detection (FFD). Although the outcome of the two algorithms is (and clearly should be) the same, their approach is completely different.

Wavefront Frontier Detection
The WFD algorithm uses a Breadth-First Search (BFS) along all open-space marked positions on the map which have not yet been considered. This will avoid a position being considered more than once while still ensuring that all frontier points will be found. A frontier point is a position on the map which is marked as unknown space and is right next to a position marked as open-space. By considering only open-space marked cells, the algorithm is guaranteed to find only direct unknown neighbors. The speed of the algorithm is determined by the size of the cells and the size of the map. Bigger cells ensures faster frontier detection but comes at the cost of lower accuracy. A bigger map requires more calculation as more open-space positions need to be considered, thereby slowing the algorithm down.

Fast Frontier Detection
Where WFD considers map positions, FFD uses the robots measurements in its calculation of the frontiers. Only newly obtained sensor data is utilized by FFD. From this laser data a so-called contour is generated. A contour is an outline of the last laser measurement made by the robot. The contour will be tested for new frontier points, which will then be added to the existing set of frontier points. This set is tested for overlaps of frontiers, which will be merged if an overlap is found. Frontier points which lie outside of the sensors range of the last measurement will be discarded.

An advantage of this approach is that the algorithm only considers the most recent sensor data, which makes the algorithm not dependent on the size of the map. However, the sensor data needs to be tightly integrated with the map to detect new frontiers, which might not be straightforward.

4.3 Exploration Strategies

Now that the frontiers are calculated, the robot needs to decide to which frontier to explore first. For this there are a number of possible strategies.

A first option would be to select the frontier which is closest to the robots current position. This strategy will lead to rapid exploration of unknown areas. However, it is an unstable approach, meaning that initial conditions can have an effect on the map layout, and, in an large environment, the robot may just explore a single corridor.

Another strategy is to select the frontier which is closest to the robots initial position. The result of this strategy is a breadth-first like search. Although the robot will traverse already explored areas more often, it will be able to correct the odometry data with the help of sensor readings in these known areas. It is a more stable approach, as in open space environments the robot will explore a growing circular region around the origin.

A final strategy would be to select the next frontier in the current route. The robot will go to the frontier which is not necessarily the closest to the robot but lays in the direction the robot is traveling in. This will lead to less turns of the robot and more straight movement. This strategy could lead to a faster traversal of certain areas but cause the robot to backtrack more than the other strategies.

4.4 Pathfinding

The last element of the exploration is a navigation algorithm. This computes an actual route for the robot to take in order to reach the selected frontier. Because implementing a pathfinding algorithm is out of the scope of this article, the ROS navigation stack was used for navigation.

The ROS navigation stack uses three inputs to generate a safe path for the robot to follow. First it uses the map supplied by the SLAM algorithm to calculate a global path from the robots position to the current goal. Additionally, the robots odometry and sensor data is utilized to update a local path in order to avoid obstacles. When the robot faces previously unseen obstacles or the global path is impossible to follow, the navigation stack will automatically re-plan. As it is a generic package meant for all types of robots, it is configured to specifically work with the TurtleBot platform.

The ROS navigation stack is a highly configurable set of tools designed for autonomous navigation based on a grid map, a so called occupancy grid. The map provided may be a static map of the environment, or dynamically created during runtime by a SLAM algorithm. When provided with a goal position and pose, it uses a global and a local planner to move the robot to the goal. On it's way, the robot will autonomously re-plan if it lost track and avoid obstacles on it's path. Given sufficient time and a possible solution, the navigation stack is claimed to be able to move the robot to a target position within reasonable accuracy.

5 Landmark Detection

Exploration strategies and SLAM algorithms provide the robot with the abilities to build a map it can navigate on. But having a navigating robot is not the end goal, because in order to serve a useful purpose, the robot should be able to navigate detect and subsequently navigate to interesting points. The points of interest (POI) might be all kinds of task-dependent waypoints, such as rocks for space probes, kitchen electronics for household robots or simply the robot's charging station. As a proof of concept, the TurtleBot equipped with a Microsoft Kinect sensor will detect chessboard patterns, and mark their positions in the map. After successful map generation, the robot is able to efficiently navigate from all explored areas to the landmarks.

ROS includes OpenCV libraries, and also provides cv_bridge, a library to convert ROS message data types to OpenCV image types. To obtain imagery from Microsofts Kinect sensor, ROS uses the OpenNI driver. OpenCV provides a predefined classifier to detect planar chessboards. The image coordinates of the chessboard midpoint are transformed into a yaw angle relative to the robot's heading. From the Kinect depth channel, the distance to the chessboard is estimated. Then, the relative position of the chessboard to the robot is converted to a global position in the map frame. ROS then publishes these occurrences as point cloud. A straightforward clustering algorithm reduces the number of points generated. Featuring a fixed maximum cluster diameter and a minimum cluster size of 1, the algorithm used is a variant of Quality Threshold clustering. More sophisticated approaches may be used instead, such as distribution-based clustering using Gaussian filters, if the algorithm shows to be ineffective.

6 Implementation

6.1 GraphSLAM

For this article a GraphSLAM algorithm is implemented. GraphSLAM is a landmark based SLAM algorithm. However, since the TurtleBot only is equipped with a laser sensor, it is difficult to distinguish between different landmarks. Instead, the algorithm can use map features like walls, chairs and other obstacles to build a constraint graph. This graph consist of poses and corresponding laser data measured at that pose. There are two possible ways to produce these constraints.

The first possibility is to use the odometry data provided by the robots odometry sensors. These sensor measure the movement of the robot after it moved. When these sensor would produce perfect data, then there would be no use for a SLAM algorithm, since the algorithm would be able to compute the exact path of the robot. However, the odometry data contains errors because of aspects like different surfaces the robot travels on or the robot getting stuck while the wheels keep turning.

Laser sensor data is the second possibility to produce and correct constraints for the graph used by GraphSLAM. Using this data, the algorithm is able to use a technique called scan matching. With this technique both old and new odometry can be corrected based on other existing constraints present in the graph.

For the data structure of the graph, a vector (ξ) containing all robot poses and a sparse matrix (Ω) containing the laser scans were used. For this, the *Eigen* library [6] is used. The matrix Ω is a $3n * 3n$ zero matrix and the vector ξ is a $3n$-dimensional zero vector, where n is the number of measured poses to be stored in the graph. During the traversal of the robot through the environment, newly obtained constraints are added to Ω and ξ depicted in Equations 4 and 5 respectively. In these equations p_i represent a three-dimensional vector $(x_i, y_i, \theta_i)^T$ containing the coordinates of the robot at time i. Every entry in Ω or ξ not depicted in Equations 4 or 5 is zero.

$$\Omega_{new} = \Omega_{old} + \begin{array}{c} \\ \vdots \\ p_i \\ \vdots \\ p_j \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} \cdots & p_i^T & \cdots & p_j^T & \cdots \end{array} \\ \begin{pmatrix} & & & & \\ & 1 & & -1 & \\ & & & & \\ & -1 & & 1 & \\ & & & & \end{pmatrix} \end{array} \tag{4}$$

$$\xi_{new} = \xi_{old} + \begin{array}{c} \vdots \\ p_i \\ \vdots \\ p_j \\ \vdots \end{array} \begin{pmatrix} \\ p_i - p_j \\ \\ p_j - p_i \\ \\ \end{pmatrix} \tag{5}$$

After adding a new constraint to the graph, the graph is evaluated. The goal for this step is to approximate the path of the robot as well as possible. For this the equation $\Omega\ \mu = \xi$ is solved for μ, where μ represents the most likely poses of the robot according to the constraints in the graph.

6.2 Filtering

Solving the linear equations in order to evaluate the graph is computationally expensive. The cost of solving this set of equations will grow relatively to the size of the graph. Therefore, not all measurements are taken into account, as the robot produces sensor data at a rate of 10 Hz. Instead, only measurements taken from a position that deviates from the last pose in the graph from a certain threshold will be added to the graph. More measurements lead to more computations but also to a better and more accurate map.

6.3 Generating a map

One of the last steps in the complete SLAM process is generating a visual representation of the environment. From the graph calculated by the GraphSLAM algorithm, an occupancy grid will be generated. An occupancy grid is a grid based map where each cell gets assigned a value. A value of -1 represents unknown space and a value of $0 - 100$ represents the probability of an obstacle being present at that location.

For this article two different map building strategies are implemented.

Triangulation-based map generation

The first strategy is a triangulation-based strategy. A simplified illustration of this is shown in Figure 3. For each pose the robot has obtained sensor data, a triangle is constructed for each pair of consecutive laser beams (A, B) and the robots pose. All cells which lie inside this triangle on the occupancy grid are marked as open space. Every cell connecting the two end points (A, B) of the two laser beams is marked as occupied. Marking a space as occupied only happens whenever the laser scan detects an object closer than the maximum range of the laser sensor.

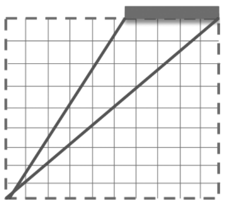

Figure 3: Triangulation-based map generation

This strategy results in maps that are not completely correct, because it is assumed that the area inside the triangle is open space and the space between the end points of the laser scans is occupied with an object. Therefore, this strategy can only be used whenever the angle between two measurements is small enough.

Another disadvantage of this strategy is that a lot of computations are needed to check which cells lie within the triangle. Each cell that lies within the bounding box of the pose and the two coordinates (the grid area in Figure 3, surrounded by a dotted line), is considered. For each cell, six cross products and three dot products need to be calculated in order to determine if a cell is within the triangle. For larger maps, these computation slow down the map generation a lot.

Although there are some disadvantages to the triangulation-based map generation, the resulting maps are very fluent because the space between two consecutive scans is always marked. Because of this, there are no jags at the edge of known areas.

Beam-based map generation

A second map generation strategy is beam-based. Instead of considering all cells within the area between the robots pose and two consecutive laser scans, only the cells along a laser scan are marked. The end point of the laser scan will be marked as occupied when it is within the maximum range of the laser sensor.

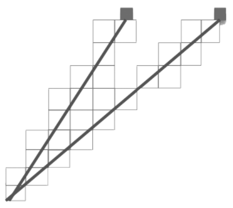

Figure 4: Beam-based map generation

This strategy is much faster as the triangulation-based strategy as there is no need for vector multiplications and fewer cells are considered to be marked. In fact, only cells that actually will be marked are considered. Also, this strategy better reflects the sensor data as it only marks map cells which are measured by the robot. As a downside the resulting maps are more jagged than the maps generated maps by the triangulation-based strategy.

6.4 Landmark detection

When the robot visits an area it has explored before, the GraphSLAM algorithm is able to improve odometry data from that area with the use of the newly obtained sensor data. In order to achieve this, a landmark detection method is required. In order to detect a landmark in the robots path, a technique called scan matching is used. Scan matching tries to find a match between two measurements taken over time. When a match is found, the algorithm is able to correct odometry data. Scan matching can be done in different ways. In this article, three different algorithms are investigated, Hill Climbing, Sampling and an iterative closest point algorithm.

ScanMatching

Scan matching tries to find a match between two measurements taken over time. When a match is found, the algorithm is able to correct odometry data. Scan matching can be done in different ways. In this article, three different algorithms are investigated, Hill Climbing, Sampling and an iterative closest point algorithm. To measure differences between two scans, one scan is mapped into the coordinate frame of the other, then, the two scans are compared ray by ray. Using a sensor model to estimate the likelihood of each ray being a correct measurement proved to be impractical, as the multiplied up probabilities quickly grow too small for common floating point variables. Instead, using a simple squared error has shown to be more robust.

Hillclimbing

Hillclimbing is a straightforward approach to find an optimum on a multidimensional fitness function. For scan matching, it is set to minimize the squared error between two scans in the two dimensional odometry model. A movement direction angle, a distance between the two scans and the rotation of the second scan are optimized. The hill climbing step size can be chosen initially and shrinks to a fraction whenever a new variable is chosen for optimization, in order to provide a result as exact as possible. As for other applications, hill climbing suffers from reaching local error minima; also, performance is dependent on the initially chose step size. A larger step size leads to a faster termination.

Particle Sampling

Instead of improving only parameter estimate, as done in hillclimbing, particle sampling produces a population of particles. Each particle represents a parameter configuration, either in Cartesian coordinates or in the odometry motion model space, and is evaluated using the squared error for a scan at that position. Iteratively, the population is resampled. Particles with lower error are chosen more often for resampling, particles with higher error will die out. New particles are sampled from a normal distribution laid around the formerly picked particle. This procedure can be repeated until all particles converged to a point or can be stopped after a fixed number of resampling procedures. Particle sampling is more robust concerning local minima, because as multiple particles are generated, the probability is higher that one reaches the descent to the global error minimum. Also, as Gaussian distributions are used, it is possible for a single particle to escape a local minimum. Robustness is dependent on finding an appropriate initial standard deviation to sample from.

Iterative Closest Point

An Iterative Closest Point Algorithm (ICP) is used to allign one so called pointcloud to another. A pointcloud is a set of vertices within a coordinate system. They represent the surface of some kind of object within the environment. In this case, map data is transformed to a pointcloud to represent the surface of the map features. It is beneficial to have a resolution as high as possible for ICP as it aims to minimize the distance between points in the pointcloud. Since the resolution of the laser sensor is high enough, ICP can be used for scan matching.

The ICP algorithm is an iterative algorithm that searches for point pairs lying close to eachother. It then iteratively rotates and translates one point cloud to fit the other as well as possible. This process continues until either a match is found or the translation is below a certain threshold, meaning a match will not be found.

The result of this iterative matching process is an alignment transform to correct one of the point clouds. This transform is then used to correct the change between two poses computed from the odometry data. This corrected pose difference is used to update one or more constraints within the graph.

7 Experiments and Results

Measuring the quality of a SLAM approach is a non-trivial task itself. Multiple features have to be considered, at least the time it took the robot to generate a complete map and the quality of the map itself. Further quality measures might be the time the robot spent *not* exploring, but moving through already known territory or being stuck, or simply how often the robot hit unexpected obstacles while navigating the finished map. SLAM also provides the pose of the robot, based on the generated map. The position prediction performance can be measured as well.

7.1 Experimental setup

In this section the setup of the conducted experiments is described. Different experiments were performed in order to find out the quality of different exploration strategies. GraphSLAM and map generation are tested in simulation by comparing the generated map with the map the simulation ran on. Scan matching noise performance was investigated separately.

7.2 Exploration Results

Exploration strategies “closest to robot” and “closest to origin” were compared. Figure 5 shows a map generated by always choosing the closest frontier point to the robot as navigation goal. Exploration was finished after 680 seconds. Figure 6 presents a map resulting from selecting frontier points closest to the origin. using this strategy, exploring the map took 1400 seconds. The time difference results from a longer path the robot travels, also, near the origin of the map, a narrow passage has to be navigated quite often. Comparing the paths of “closest to origin” in Figure 7 with the “closest to robot” path in Figure 8 shows why the latter explores much faster.

7.3 GraphSLAM experiments

Testing GraphSLAM was done qualitatively, by comparing the generated results to the simulation source. The scan matching routine of GraphSLAM was tested on different odometry noise levels.

Scan Matching Results

Matching erroneous laser scans has proven to be a non-trivial task. To investigate the performance of scan matching, two identical lasers scans were produced in Stage, noise was added to the system and the scans were matched. Noise was added as odometry noise, that is, the odometry estimate for the second scan was altered, as well as sensor noise. The process was repeated for a number of scans, generated at random positions all over the map.

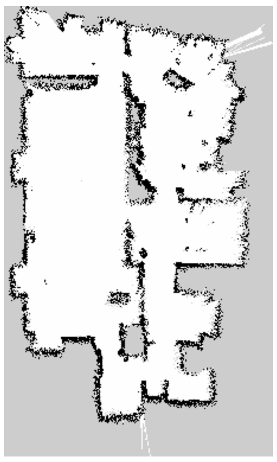

Figure 5: Generated map navigating to closest frontiers to the robot

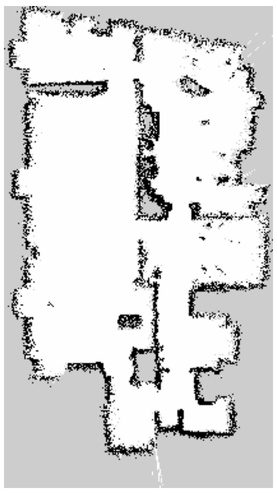

Figure 6: Generated map navigating to closest frontiers to the origin

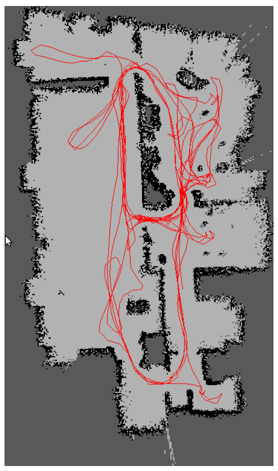

Figure 7: The robot path when selecting frontier points closest to the origin

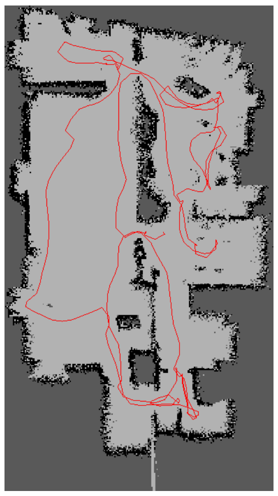

Figure 8: The robot path when selecting frontier points closest to the robot

Odometry noise

Odometry noise affects the initial error for two scans to be matched. The two scans that originally were produced at the same location are artificially expected to be a certain distance apart, to model odometry noise. The resulting error for moved distances in meters is plotted in Figure 9.

Rotational estimation error leads to a starting error shown in Figure 10. Note the plateau regions around 2.5 radians, which pose a serious problem to optimization algorithms.

Note that although the functions seem to be very smooth, the error function for single scans is by far more complicated to optimize on. See Figure 11 - it features many local minima as well as plateaus. Also, the descent to the optimum is quite steep, so that less sophisticated optimizers might step over it.

Noise Influence

In order to produce most realistic simulations, odometry and sensor noise was added to one scan. For raising noise levels, the different scan matchers were tested. The quality of a scan matching was achieved by calculating the error before and after the matching. One minus the error level after matching divided by error before matching can then be seen as "error elimination quotient".

Figure 15 shows the error elimination quotient for hillclimbing. A 95% reduction rate is only given for extremely low noise levels. When the input noise level hits 1, no correction at all can be made. The notch for very low input noises can be explained by the minimum step width of the hill climber, which seems to bee too large for that amounts of noise, so that it immediately steps over the ideal solution, and might get stuck afterwards. Generating these results, that is, matching 100 scans 315

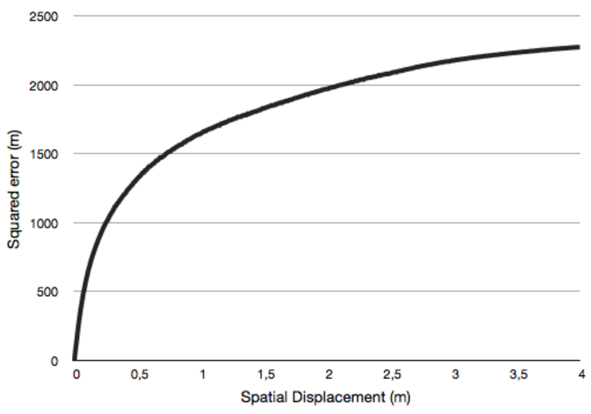

Figure 9: Squared error plotted for certain spatial displacements of scans, ranging from no displacement up to 4 meters. Averaged from 100 scans.

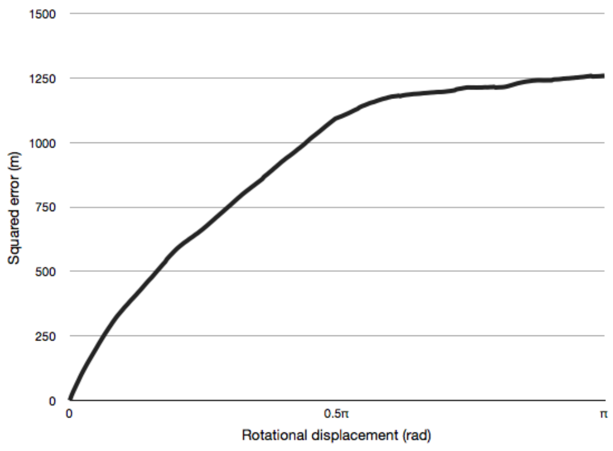

Figure 10: Squared error levels for rotational displacement of two scans, ranging from no rotation to PI radians. Averaged from 100 scans.

times took 441 seconds.

Particle sampling even performs worse than hill-climbing, most likely due to the fact that the initial particle deviation was set too low. As Figure 17 shows in detail, already on a noise level of 0.25 it does not improve the matching any more. Note however, that particle sampling will by itself return the odometry estimate instead of a matching if the input scans do not fulfill certain conditions - a minimum number of scans must be non-max readings, also, a scan with too many close-to-zero readings is rejected. As the scans are drawn randomly, it is very well possible that many of them are defunct according to these specifications. Also, rejected scans all have an equally bad score, hindering quick and early convergence, slowing the algorithm down. Generating these results, that is, matching 100 scans 315 times

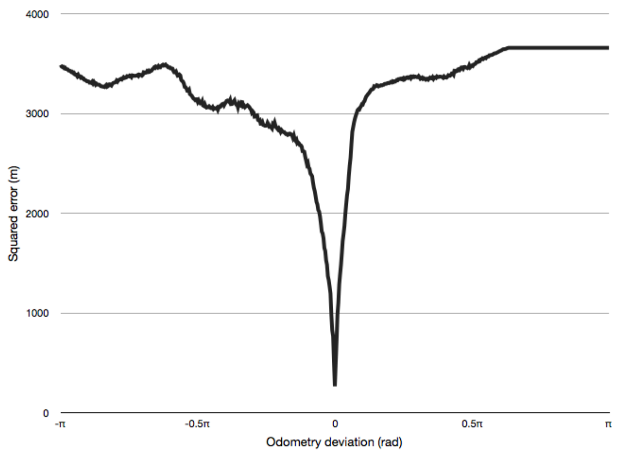

Figure 11: Squared error levels for a single scan. Error is plotted against rotational deviations from -PI to PI radians

took 918 seconds, making it the worst performing algorithm in both speed and precision.

Figure 16 shows the results of ICP. Again, the reached quality drops off quicker than for hill climbing, but ICP only took 99 seconds to match the 31500 scans, which means one could add more iterations or relax the outlier threshold in order to achieve better results. Performance-wise this is the only algorithm usable on the TurtleBot, as it has very limited processing power.

Map Generation Results

Using a simulated map in stage it is possible to compare the generated map with the underlying one. The map of the SwarmLab (Figure 2) was used for testing.

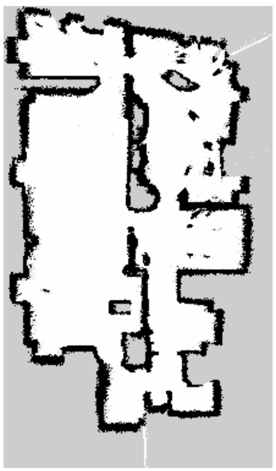

Figure 12: Generated map with activated sensor noise and the GraphSLAM algorithm, resemblance 92.95%

Figure 12 shows the resulting map generated by GraphSLAM and the triangle fill map generation

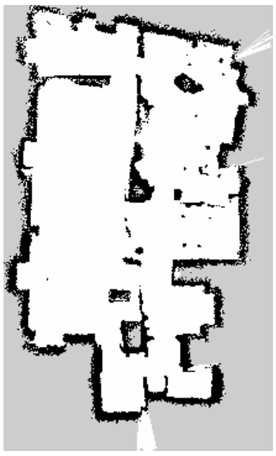

Figure 13: Generated map using ROS GMapping-SLAM, sensor noise enabled, resemblance 94.08%

method. As comparison, Figure 13 shows a map generated by the ROS GMapping-SLAM package. The resemblance to the actual map used for simulation 2, is quite high with 92.95% and 94.08%, respectively.

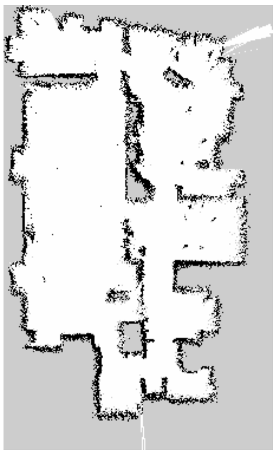

Figure 14: Map using Beam-based map generation, resemblance 90.62%

Switching the map generator to beam-based slightly lowers the resemblance to 90.62%, yielding a map as presented in Figure 14. However, this method is significantly faster and the resemblance is still acceptably high, so it is the method of choice to run on the TurtleBot.

After the images are positioned, the differences between both maps are evaluated on pixel level. Each wall pixel that is placed correctly is weighted against each wall pixel placed incorrectly, and each open space pixel is weighted against each open space pixel placed in a wall or otherwise inaccessible area. The remaining ratio represents an overall fitness between 0 and 1. However, the algorithm relies on control points found by SIFT, lower results than 0.8 are seldom generated, as for poor quality maps, SIFT does not enough control points, and for poorly fitting maps, RANSAC filters out too many control points to perform a value optimization. These problems make further evaluations for low quality maps impossible. For high quality maps, the algorithm provides a good estimate for map quality.

8 Conclusions & Future Research

The Robot Operating System proves to be a well-design framework for working with the TurtleBot platform. ROS provides libraries for the most common actions and accessories, such as the Hokuyo laser scanner and the Microsoft Kinect camera and provides the foundation to implement a wide variety of algorithms. Because ROS is designed to be suitable on all kinds and types of robot, extensive research is required to achieve the desired results.

In order to explore an environment autonomously, a frontier detection algorithm as well as an exploration strategy was successfully implemented. The frontier detection calculates the borders between the known and unknown fast enough for the exploration strategy to select a new goal when the robot reaches a frontier. Selecting the frontier closest to the robot proved to be the fastest exploration strategy, since other caused the robot to backtrack more often.

GraphSLAM is an intuitive approach to SLAM and works well with relative low odometry noise. It benefits from having easily distinguishable areas or landmarks, in order to correct the odometry errors. Odometry data as initial estimate for scan matching was not precise enough to yield satisfactory results in a real life environment. A way to circumvent the occurring problems might be to register all scans between two graph slam nodes, and apply a Kalman filter on them before scan matching. This should improve the odometry estimate and lower the sensor noise effects. More sophisticated scan matching variants like Iterative Closest Line or Correlative Scan Matching as proposed by Olson [10], can be investigated.

For generating the map based on the graph produced by the GraphSLAM algorithm, two map generation strategies were implemented. Where the triangulation-based strategies results in smooth maps, it is very resource-intensive. The beam-based strategy produced the map a lot faster, since it has to consider a lot fewer grid cells but only considers the cells which will be marked by the current laser scan.

As object detection is one of the main features of OpenCV and it provides a prebuilt classifier for chessboard detection, identifying and locating it in an image was a straightforward task. ROS provides the necessary tools to be able to access and convert Kinect imagery to OpenCV image data easily. However, generating a stereo vision image takes up nearly all processing power of the TurtleBot, making it infeasible to run navigation, stereo imaging and object detection simultaneously. Another bottleneck is the available network, over which the image data is streamed, making it probable that during movement, chessboards in sight of the robot can remain undetected.

References

[1] Aghamohammadi, A.A., Taghirad, H. D., Tamjidi, A. H., and Mihankhah, E. (2007). Feature-Based Laser Scan Matching For Accurate and High Speed Mobile Robot Localization. *EMCR'07.*

[2] Bailey, T. and Durrant-Whyte, H. (2006). Simultaneous Localisation and Mapping (SLAM): Part II State of the Art.

[3] Besl, P.J. and McKay, N.D. (1992). A Method for Registration of 3-D Shapes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Vol. 14, No. 2, pp. 239–256.

[4] Bosse, M.C. (2004). ATLAS: a framework for large scale automated mapping and localization. *Ph.D. dissertation, Massachusetts Institute of Technol- ogy, Cambridge, MA, USA.*

[5] Grisetti, G., Kümmerle, R., Stachniss, C., and Burgard, W. (2010). A Tutorial on Graph-Based SLAM. *Intelligent Transportation Systems Magazine, IEEE*, Vol. 2, No. 4, pp. 31–43.

[6] Jacob, B. and Guennebaud, G. (2013). Eigen, a C++ library for linear algebra. `http://eigen.tuxfamily.org`.

[7] Keidar, M. and Kaminka, G.A. (2012). Robot exploration with fast frontier detection: theory and experiments. *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, Vol. 1, No. 2, pp. 113–120.

[8] Lee, C.S. (2010). A Review of Submapping SLAM techniques. *A Thesis Submitted for the Degree of MSc Erasmus Mundus in Vision and Robotics (VIBOT).*

[9] Olson, E. (2008). Robust and efficient robotic mapping. *Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA.*

[10] Olson, E.B. (2009). Real-time correlative scan matching. *PRobotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pp. 4387–4393.

[11] Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A.Y. (2009). Ros: an open-source robot operating system.

[12] (2013). Robot operating system. `http://www.ros.org`.

[13] Thrun, S. and Montemerlo, M. (2006). The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures. *The International Journal of Robotics Research*, Vol. 25, No. 5-6, pp. 403–429.

[14] Thrun, S., Burgard, W., and Fox, D. (2006). *Probabilistic Robotics.* MIT Press.

[15] (2013). Turtlebot platform. `http://turtlebot.com`.

[16] Wullschleger, F.H., Arras, K.O., and Vestli, S.J. (1999). A Flexible Exploration Framework for Map Building. *Advanced Mobile Robots, 1999. (Eurobot '99) 1999 Third European Workshop on*, pp. 49 –56.

[17] Yamauchi, B. (1997). A Frontier-Based Approach for Autonomous Exploration. *Proceedings of CIRA'97, July 10-11, 1997 in Monterey, California, USA.*

A Results

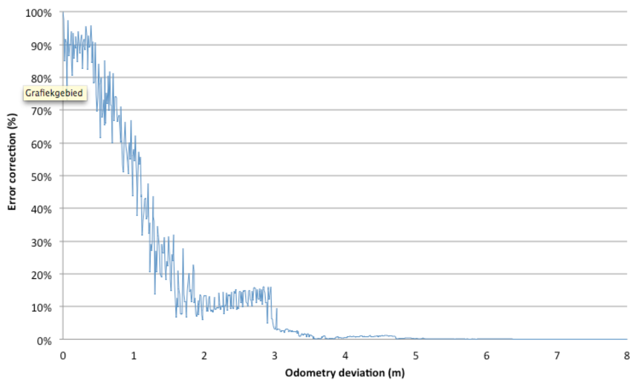

Figure 15: Error correction rate for odometry estimation errors using hill climbing

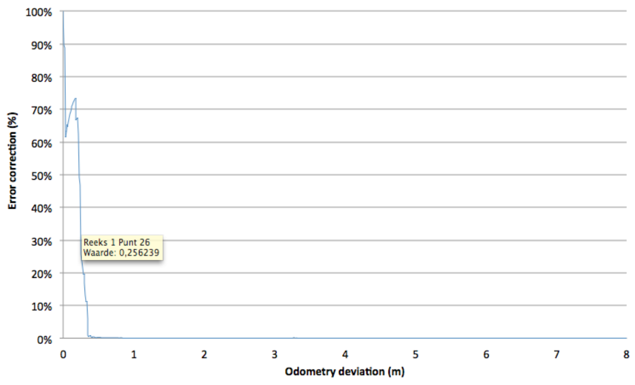

Figure 16: Error correction rate for odometry estimation errors using ICP

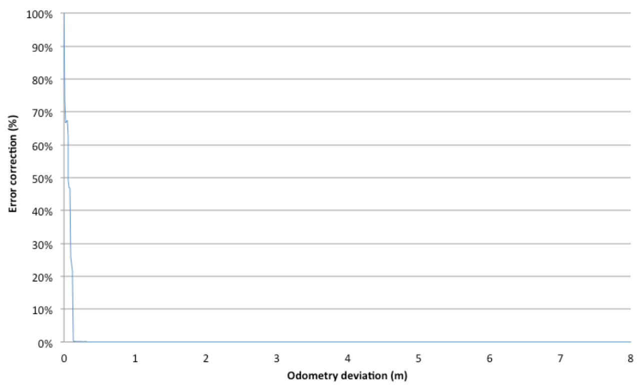

Figure 17: Error correction rate for odometry estimation errors using particle sampling

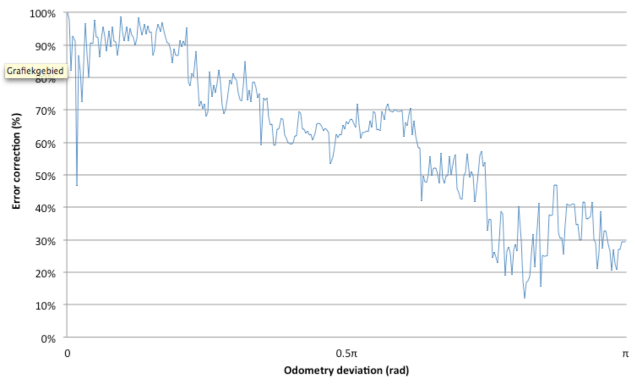

Figure 18: Error correction rate for rotational odometry estimation errors using hill climbing

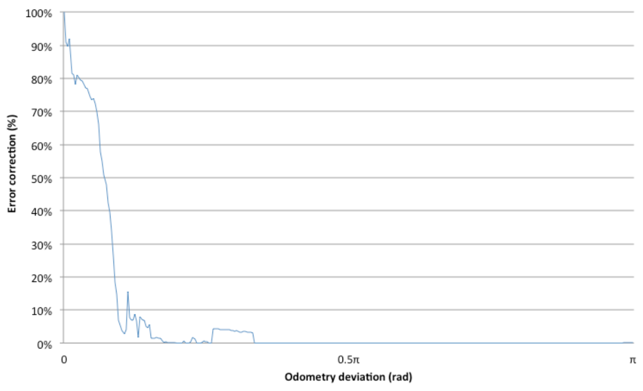

Figure 19: Error correction rate for rotational odometry estimation errors using ICP

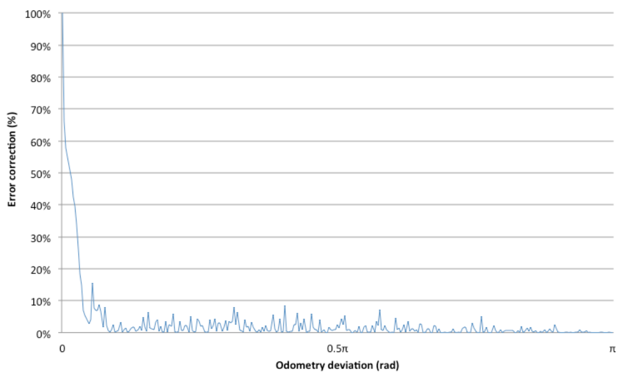

Figure 20: Error correction rate for rotational odometry estimation errors using particle sampling

B Results loop closing

An example for the improvement of the map using loop closing is given by the following images. The dark-colored path is the true path the robot traveled, the lighter one is the believed path. In the first image it has a little offset and the wall in the center does not fit perfectly. In the second image the robot has revisited a earlier pose and detected a loop closing. The believed path is corrected and less offset, the wall in the center is consistent.

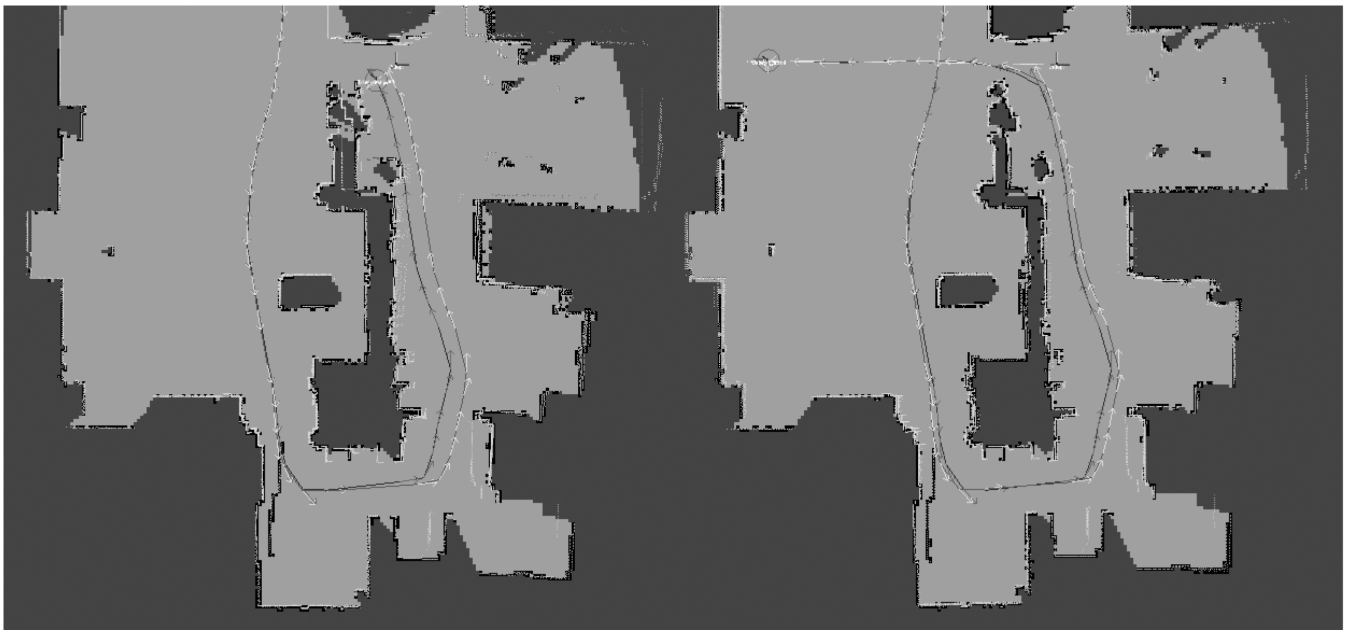

Figure 21: An example of loop closing